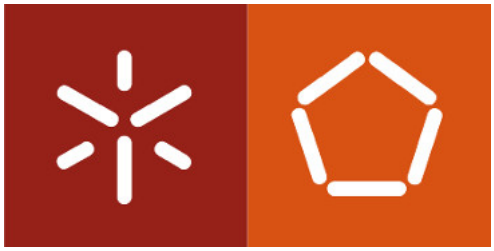


Universidade do Minho
Escola de Engenharia
Mestrado em Engenharia Informática

Unidades Curricularres:
Paradigmas de Sistemas Distribuídos
Sistemas Distribuídos em Grande Escala

Ano Letivo de 2023/2024

Serviço de Partilha de Álbuns

Grupo:

PG53816	Francisca Quintas Monteiro de Barros
A91775	José Pedro Batista Fonte
PG53985	José Rafael Cruz Ferreira
PG52698	Nuno Ricardo Oliveira Costa

19 de junho de 2024

1 Introdução

No âmbito das unidades curriculares de Paradigmas de Sistemas Distribuídos e Sistemas Distribuídos em Grande Escala, foi-nos proposto implementar um serviço de "Partilha de Álbuns"com as seguintes condições:

- Os ficheiros podem ser **fotografias** ou **vídeos**
- Deve permitir **gerir utilizadores** e **associar utilizadores a albúns**
- Deve responder às operações de **upload** e de **download** de ficheiros dos álbuns;
- Deve permitir **a classificação dos ficheiros**;
- Deve permitir ainda **discussão em tempo real** entre utilizadores para cada álbum;

O serviço é dividido em três entidades: **Clientes**, **Servidor Central** e **Servidores de Dados**. Para utilizar o serviço, qualquer cliente procede à autenticação e, posteriormente, estabelece ligação com o Servidor Central pela qual poderá administrar operações fornecidas pelo serviço tais como a criação e edição de albúns. Além disto, o Servidor Central é utilizado para obtenção dos metadados de um albúm para uso local por parte dos clientes que lhe solicitem. Os Servidores de Dados têm como tarefa principal o armazenamento o conteúdo dos ficheiros através de uma estrutura de content-addressable storage distribuída. Neste serviço operações como edição de um albúm por múltiplos clientes assim como a discussão em tempo-real, acima mencionada, é feita através de conexões peer-to-peer entre os utilizadores envolvidos nas mesmas.

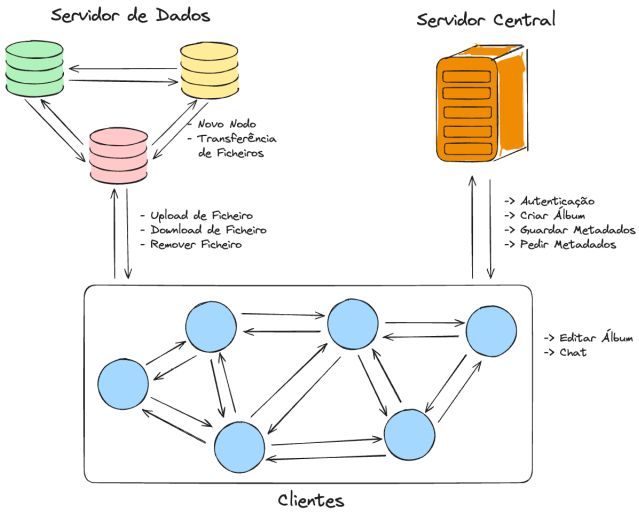


Figura 1: Arquitetura do Sistema

2 Cliente

Tal como foi referido, o cliente inicialmente tem que fazer a autenticação, logo são lhe apresentadas duas opções iniciais: registo ou autenticação. Após o preenchimento das credenciais, esta informação

é enviada para o servidor central que faz a devida validação e responde se o pedido teve sucesso ou não. Caso não tenha sucesso, simplesmente apresenta as mesmas opções novamente, caso contrário, tanto no registo como na autenticação, o cliente fica autenticado e é lhe apresentado um menu com as seguintes opções:

- Create Album
- Edit Album
- Request Album Metadata
- Request File Data
- Logout

2.1 Criar álbum

O utilizador simplesmente tem de inserir o nome do álbum que quer criar, envia para o servidor central que verifica se o nome é válido. Se for o álbum fica criado, com nenhum ficheiro e apenas um utilizador associado - o utilizador que criou o álbum.

2.2 Sessão de edição do álbum

Esta é a operação mais importante do lado do cliente, visto que é aqui que são criadas sessões de edição dos álbuns que, além de permitirem alterar o conteúdo do álbum disponibilizam também um serviço de chat. Estas duas funcionalidades funcionam num modelo peer-to-peer, e apenas quando o último utilizador sai da sessão, é que as alterações são tornadas efetivas no servidor central.

O primeiro passo a entrar numa sessão de edição é mandar um pedido ao servidor a avisar que quer entrar na sessão e a pedir os metadados daquele álbum. Se o utlizador tiver as permissões necessárias, é lhe enviado o conteúdo do álbum e este está pronto a mandar pedidos e alterar o álbum.

Como apenas foi feita uma interface simples de texto, a estratégia seguida para enviar mensagens no chat e conseguir editar o álbum é a seguinte: todas as mensagens começadas por \ são consideradas comandos para editar o album e tudo o resto são mensagens enviadas para o chat.

Tanto o serviço de chat como o de edição do álbum usam o ZeroMQ, mais especificamente o padrão publisher-subscriber - com o uso de sockets XPUB e XSUB - que tiram partido de um broker que é intermediário para a troca de mensagens entre clientes. Desta forma existe uma abstração do cliente em relação aos outros utilizadores, não precisa de saber as suas informações, simplesmente quem entra na sessão subscreve ao conteúdo dessa sessão e quando quer enviar algo publica, tornando o processo mais simples e intuitivo.

Todas as mensagens enviadas começam com um identificador do álbum que serve como filtro para a subscrição dos clientes, desta forma estes apenas recebem mensagens destinadas à sessão em que se encontram.

Para identificador do utilizador na sessão inicialmente usámos o pid, no entanto, isto dava conflito quando o utilizador entrava e saia de uma mesma sessão, porque as suas estruturas eram inicializadas

novamente, no entanto, era considerado um mesmo processo. Logo o identificador passou a ser uma junção do pid com um valor que incrementa a cada sessão que o utlizador se junta.

2.2.1 Chat

O serviço de chat apenas se preocupa com a entrega causal das mensagens, logo cada cliente tem um *VersionVector* que é atualizado a cada mensagem que envia e recebe.

As mensagens enviadas para o socket são deste tipo: album_name:"chat":pid:vv:mensagem

Onde o album_name é o nome do álbum cujo o utilizador está a editar, pid é o pid do processo que está a enviar a mensagem, vv é o version vector atual dele e a mensagem a mensagem que ele quer enviar para o chat.

Quando um cliente recebe uma mensagem deste tipo dá parse do VersionVector e verifica se ela pode ser entregue:

$$V[j] + 1 = V_m[j] \wedge \forall k \neq j \cdot V_m[k] \leq V[k]$$

Se puder, imprime no ecrã, caso contrário é adicionada às mensagens em espera.

Sempre que uma mensagem é entregue, verifica-se se alguma das mensagens à espera podem ser também entregues.

Nota: A primeira mensagem, visto que os utilizadores podem entrar a meio da sessão, considera-se que pode sempre ser entregue imediatamente, atualizando-se o version vector para o que recebeu.

Posteriormente, foram feitas otimizações no envio dos version vectors, para enviar apenas as dependências necessárias (dependências diretas):

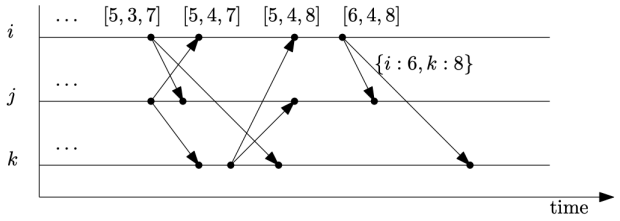


Figura 2: Otimização - Casual Broadcast

2.2.2 Editar o álbum

No inicio da sessão, o cliente pede ao servidor central os metadados para aquele álbum e com esta informação inicializa o seu estado.

Optámos por CRDTs de estado, visto que os utilizadores de uma sessão podem entrar a meio e se usássemos CRDTs de operações era necessário ter um maior controlo e ir guardando a lista de todas

as operações e quando entrasse um nodo novo este teria de as receber todas. Como queríamos um bocado abstrair da entrada e saída de nodos, optámos então por CRDTs de estado.

Estes, no entanto, são bastante mais pesados na rede, no entanto como só são trocados entre utilizadores de uma sessão considerámos que isto não seria um problema.

Tentámos procurar outras soluções, como Delta-State CRDTs, e começámos a tentar implementá-los, no entanto, não houve tempo para concluir esta implementação, baseada neste paper - Efficient State-based CRDTs by Delta-Mutation.

$$
\begin{aligned}
\Sigma &= \mathcal{P}(\mathbb{I} \times \mathbb{N} \times E) \times \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\sigma_i^0 &= (\{\}, \{\}) \\
\mathsf{add}_i^\delta(e, (s, c)) &= (\{(i, n, e)\}, \{(i, n+1)\}) \\
&\quad \text{with } n = \mathsf{max}(\{k \mid (i, k) \in c\}) \\
\mathsf{rmv}_i^\delta(e, (s, c)) &= (\{\}, \{(j, n) \mid (j, n, e) \in s\}) \\
\mathsf{elements}_i((s, c)) &= \{e \mid (j, n, e) \in s\} \\
(s, c) \sqcup (s', c') &= ((s \cap s') \cup \{(i, n, e) \in s \mid (i, n) \notin c'\} \\
&\quad \cup \{(i, n, e) \in s' \mid (i, n) \notin c\}, c \cup c')
\end{aligned}
$$

Figura 3: Add-wins observed-remove Delta-CRDT set, replica i.

O estado de cada cliente é constituído então por dois Causal CRDTs do tipo ORSet (Observed-remove set): utlizadores e ficheiros.

$$
\begin{aligned}
\mathsf{ORSet}\langle E \rangle &= \mathsf{Causal}\langle \mathsf{DotMap}\langle E, \mathsf{DotSet} \rangle \rangle \\
\mathsf{add}_i(e, (m, c)) &= (m\{e \mapsto \{(i, c[i]+1)\}\}, c\{i \mapsto c[i]+1\}) \\
\mathsf{remove}_i(e, (m, c)) &= (\{e\} \lhd m, c) \\
\mathsf{elements}((m, c)) &= \mathsf{dom}\, m
\end{aligned}
$$

Figura 4: ORSet operations

Sempre que uma operação é executada o novo estado é enviado e todos os nodos atualizam a sua réplica local (para não ser tão pesado na rede, esta atualização poderia apenas ser feita de x em x tempo).

Quando um nó sai, envia o seu estado e sai da sessão, enviando ao servidor central, o seu estado final. O servidor central, se este for o último utilizador na sessão, atualiza os seus dados com as alterações feitas.

A única estratégia que falta referir é a forma como lidamos com os ratings dos ficheiros. Como no enunciado dizia que ao consultar um álbum, um utilizador apenas recebe os ficheiros e a média das pontuações do mesmo, considerámos que cada utilizador apenas tem acesso às suas pontuações, logo esta informação é guardada localmente por cada um e alterada durante a sessão localmente. No final, tal como foi referido, o estado é enviado para o servidor central que guarda a informação das pontuações para mais tarde adicionar (caso não seja o último da sessão ainda).

2.3 Pedir Metadados de um Álbum

O utilizador insere o nome do álbum que pretende consultar e recebe a lista de ficheiros desse álbum, com nome e média das pontuações dadas pelos utilizadores.

2.4 Download de ficheiro

Para o download de um ficheiro é necessário especificar o nome do álbum e do ficheiro, o servidor central faz a verificação das permissões daquele utilizador (verifica se ele está associado àquele álbum e que o ficheiro existe) e o download é feito.

3 Servidor Central

O servidor central é feito em Erlang e cria um ator para gerir cada cliente. Devido às suas propriedades, este método não é custoso em sistemas de larga escala, visto que, ao contrário da criação de threads e processos, e apesar de terem propriedades semelhantes, estes são entidades bastante "leves".

Optámos por separar a lógica do servidor em 4 módulos essenciais: login_manager, user_manager, file_manager e session_manager.

3.1 Login_manager

Responsável pela autenticação dos utilizadores, contém um map que associa o username à password. As operações principais usadas são: criar conta, fazer login e logout.

3.2 User_manager

Responsável por associar utilizadores a álbuns, contém um map que associa o nome de um álbum a um set de utilizadores. Este módulo é maioritariamente utilizado para verificar se um utilizador pode editar um álbum.

3.3 File_manager

Responsável por associar ficheiros e as respetivas classificações a álbuns, contém um map com esta estrutura: `{Album : {Utilizador : Rating}}`.

3.4 Session_manager

Responsável por gerir os utilizadores e as classificações atribuídas aos ficheiros durante uma sessão de edição de um álbum, permitindo guardar os vários estados de um álbum durante a edição e apenas aplicar essas alterações quando o último utilizador abandona a sessão.

4 Servidor de Dados

Neste serviço, o servidor de dados desempenha a função principal de persistência de dados de forma escalável e distribuída. Para alcançar esse objetivo, o servidor de dados é implementado seguindo a arquitetura de uma Tabela de Hash Distribuída (DHT) com Consistent Hashing, cujo funcionamento será explicado na secção seguinte.

É importante denotar que, nesta implementação, o servidor de dados não assume falhas e não inclui controlo de sessão, que é garantido pelo servidor central. Portanto, um cliente com uma chave válida pode aceder a um arquivo armazenado.

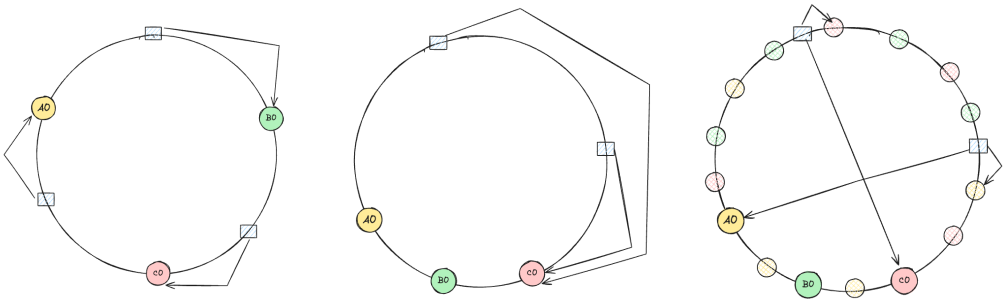
4.1 DHT com Consistent Hashing

Uma DHT utiliza o mesmo método de armazenamento de um Tabela de Hash, porém divide um espaço N de chaves entre vários nós, fisicamente separados, o que proporciona maior resiliência e desempenho do servidor.

Existem várias abordagens para implementar uma DHT, mas o grupo optou por utilizar o Consistent Hashing, devido à sua menor complexidade de implementação em comparação com Chord ou Kademlia. Esta escolha também é a mais aconselhada para serviços de pequena e média dimensão. O Consistent Hashing permite um bom balanceamento de carga entre os nós, poucas transferências de chaves entre eles, uma procura com complexidade O(1) e um tamanho de estado de O(N).

Funcionamento Básico

O funcionamento básico de uma DHT com Consistent Hashing consiste na distribuição de chaves e valores entre os nós da rede, como mostra a figura 5 a). No entanto, pode surgir um problema de balanceamento de carga devido à imprevisibilidade das posições dos nós na rede, como ilustra a figura 5 b) . Uma solução eficiente para este problema é a introdução de Nodos Virtuais, que não existem fisicamente, mas melhoram a distribuição dos nós físicos na DHT, como se verifica na figura 5 c), corrigindo assim o desequilíbrio na distribuição de chaves.



(a) Ideal (b) Não Balanceada (c) c/Nodos Virtuais

Figura 5: Distributed Hash Table

4.2 Implementação

O Servidor de Dados é implementado utilizando Java com a biblioteca de Reactive gRPC, assim sendo, é necessário especificar o serviço a ser utilizado nos nodos do servidor.

O serviço é denominado DataServerNode e define os métodos gRPC e mensagens disponíveis para interação entre os nós e com clientes. Inclui operações para adicionar um novo nó, fazer download e upload de ficheiros, remover ficheiros, e verificar a disponibilidade de um nó.

NodeServer

A classe NodeServer implementa o serviço de cada nó do sistema distribuído. Ele utiliza gRPC para comunicar entre nós e com clientes, e oferece todas funcionalidades descritas.

Métodos Principais:

- **start(ArrayList<Pair<String, String>> neighborNodes)**: Inicia o servidor, adiciona os seus nós virtuais ao anel, informa os nós vizinhos, adiciona-os caso respondam, e abre um socket de servidor para aceitar novos pedidos.
- **addNode(String ip_add, String ip_port)**: Adiciona um novo nó ao anel, criando todos os nodos virtuais.
- **lookupClosestNode(byte[] key)**: Procura o nó mais próximo no anel com base numa chave.
- **migrateKeys() e migrateFileTransfer()** : método para migrar todos os ficheiros, quando um novo nodo é adicionado à DHT. Analisa todas as chaves guardadas na sua *storage*, calcula se pertence ao seu nodo, caso não pertença, transfere para o nodo vizinho correto e apaga o ficheiro no seu armazenamento.

Métodos gRPC:

- **newNode(Single<NewNodeRequest> request)**: Adiciona um novo nó ao anel quando solicitado por outro nó e migra todas as chaves, caso necessário.
- **downloadFile(Single<DownloadFileRequest> request)**: Responde a pedidos de download. Se a chave pertencer ao nodo, confirma. Caso contrário, indica o nodo certo.
- **downloadFileTransfer(Single<DownloadFileRequest> request)**: Responde ao cliente com uma stream de *chunks* do ficheiro pedido.
- **uploadFile(Single<UploadFileRequest> request)**: Responde a pedidos de upload. Se a chave pertencer ao nodo, confirma. Caso contrário, indica o nodo certo.
- **uploadFileTransfer(Flowable<UploadFileRequestTransfer> request)**: Recebe uma stream de *chunks* de um ficheiro. Utiliza um buffer, e só escreve para ficheiro a cada N *chunks*.
- **removeFile(Single<RemoveRequest> request)**: Remove um arquivo do armazenamento.
- **ping(Single<PingRequest> request)**: Testa a conexão com um nó.

NodeState.java A classe NodeState representa o estado de um NodeServer no sistema distribuído. Ela mantém informações como endereço IP, porta, chave única, o estado da DHT (ring) e o seu armazenamento(storage).

VirtualNode.java A classe é utilizada para representar um nodo virtual no ring de um NodeState. Cada nó virtual está associado a um nó físico e tem um índice de réplica para distinguir entre múltiplas réplicas do mesmo nó físico. Utiliza uma função de hash para gerar chaves únicas para cada nó virtual, com base na chave do nó físico e no índice de réplica.

HashFunction.java A classe fornece métodos para gerar e manipular hashes usando o algoritmo SHA-256. O método **generateHash** recebe uma string original e retorna o seu hash em formato de bytes. A classe métodos é util para calcular todas as chaves utilizadas na DHT.

5 Conclusão

Este projeto, que englobou diferentes vertentes de investigação e aprofundamento dos tópicos abordados nas aulas de ambas as unidades curriculares, exigiu uma distribuição estratégica de tarefas. A necessidade de fragmentar as funcionalidades deste sistema de grande escala em implementações mais granulares tornou-se evidente e o uso de tecnologias que simplificaram a complexidade das redes de comunicação foi fundamental para a implementação destes serviços. Estes incluem a gestão eficiente e robusta da transmissão de dados entre diferentes componentes, bem como o processamento cuidadoso dos dados pelos próprios nodos, tendo em conta a concorrência destes sistemas. Para suportar estas especificações é crucial uma arquitetura algorítmica pensada para gerir, de forma resiliente e robusta, a capacidade de resposta que os elementos do sistema conseguem dar ao cliente.